LightCross: Sharding with Lightweight Cross-Shard Execution for Smart Contracts

Xiaodong Qi
Nanyang Technological University
xiaodong.qi@ntu.edu.sg

Yi Li
Nanyang Technological University
yi_li@ntu.edu.sg

***Abstract*—Sharding is a prevailing solution to enhance the scalability of current blockchain systems. However, the cross-shard commit protocols adopted in these systems to commit cross-shard transactions commonly incur multi-round shard-to-shard communication, leading to low performance. Furthermore, most solutions only focus on simple transfer transactions without supporting complex smart contracts, preventing sharding from widespread applications. In this paper, we propose LightCross, a novel sharding blockchain system that enables efficient execution of complex cross-shard smart contracts. First, LightCross offloads the execution of cross-shard transactions into off-chain executors equipped with the TEE hardware, which can accommodate execution for arbitrarily complex contracts. Second, we design a lightweight cross-shard commit protocol to commit cross-shard transactions without multi-round shard-to-shard communication between shards. Last, LightCross lowers the cross-shard transaction ratio by dynamically changing the distribution of contracts according to historical transactions. We implemented the LightCross prototype based on the FISCO-BCOS project and evaluated it in real-world blockchain environments, showing that LightCross can achieve 2.6× more throughput than state-of-the-art sharding systems.**

***Index Terms*—sharding, blockchain, smart contract**

I. Introduction

Sharding is considered as a prominent approach to scale blockchains [1, 2]. The key idea is to split the blockchain system into numerous small groups, called *shards*, to handle different sets of transactions in parallel and maintain different parts of the whole ledger separately [3]. Due to the separation of ledgers, a critical challenge for sharding systems is to handle *cross-shard transactions* (CSTx), i.e., transactions that access data managed by different shards, requiring a so-called *cross-shard commit protocol* established among the involved shards.

Many previous sharding solutions [4–8] focus exclusively on the simplest kind of cross-shard transfer transactions, i.e., user-to-user transfers of digital funds, while ignoring how to support contract transactions efficiently. A regular transfer can be split into several *withdraw* and *deposit* operations, which can be executed at different shards independently. Yet, processing smart contract transactions is much more challenging. This is because executing a contract transaction may involve multiple complex contracts—states of different contracts need to be accessed, so it is difficult to separate a contract transaction into independent operations. Although some sharding blockchains [9–11] attempt to handle contract CSTxs, they all come with complex, multi-round and cross-shard commit protocols to deal with CSTxs, causing performance degradation in terms of throughput, latency, and scalability.

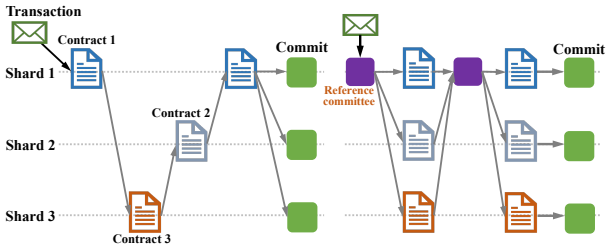


Fig. 1: Illustration of two kinds of basic protocols to process CSTxs in existing systems.

Essentially, to complete the execution of a CSTx, all the involved shards need to execute multiple invoked contracts in a certain order, e.g., RingBFT [11] and OmniLedger [4] (left-hand side of Fig. 1), or assign a special *reference committee* to dispatch and collect data or messages, e.g., AHL [10] and ByShard [12] (right-hand side of Fig. 1). However, both kinds of solutions need shard-to-shard communication between shards to coordinate the execution. This communication, rather than point-to-point communication, is much more costly, even resulting in more than 50% throughput degradation [13, 14]. Meanwhile, all accessed states should be locked for the whole protocol cycle to ensure the atomicity of CSTxs. Furthermore, some solutions just works for the simple contract transactions. For example, AHL and ByShard require the logic of a contract transaction can be properly expressed by multiple independent code fragments, which is unrealistic for complex contracts.

In blockchain sharding solutions [4, 9, 10, 15], the presence of CSTxs is inevitable. As shard number rises, the chance of requiring a CSTx increases (even over 90% [2] on a 10-shard blockchain). With a high CSTx ratio, the heavy cross-shard commit protocol may offset the benefits brought by the sharding design. Therefore, lowering the CSTx ratio is critical to an efficient sharding system. OptChain [2] attempts to relieve this issue by carefully dispatching every transaction to an optimal shard. Unfortunately, OptChain is designed toward the UTXO-based model.

Our Solution. In this paper, we propose LightCross, a novel sharding framework with a lightweight cross-shard commit protocol and low CSTx ratio. First, LightCross leverages the *trusted execution environment* (TEE) to execute CSTxs off-chain with no need of multi-round shard-to-shard communi-

cation. In particular, every CSTx is executed in the TEE at a particular node, called *executor*. On-chain shards can verify the execution results via the provided attestation mechanism [16] without re-executing it. During the execution, TEE synchronizes necessary state data from involved shards via point-to-point communication. As all the data and codes are centralized at a executor, the logic of every smart contract transaction can be arbitrarily complex.

Second, LightCross explores a lightweight cross-shard commit protocol to handle CSTxs. LightCross allows all executors to execute CSTxs speculatively and concurrently against certain contract state snapshots and then serializes these CSTxs. During the protocol, accessed states are only locked for a very short period, and all shards can still process other intra-shard transactions in most cases. LightCross amortizes the overhead of each CSTx by handling CSTxs in batches, while current solutions have to initiate a protocol instance per CSTx.

Finally, LightCross also lowers the CSTx ratio to improve the performance. Intuitively, LightCross periodically migrates smart contracts to gather contracts often invoked simultaneously to the same shard, based on knowledge learned from historical transactions. Particularly, LightCross models the relationship among contracts as a *transaction call graph*, then convert the contract migration to a graph partition problem. Then, a protocol is used to migrate the code and storage associated with contracts between all shards to lower CSTx ratio.

Our contributions can be summarized as follows:

- We propose a novel blockchain sharding framework, LightCross, based on TEE to accelerate the execution of complex contract CSTxs. LightCross adopts a lightweight cross-shard commit protocol to commit CSTxs, without requiring multi-round communication for each CSTx.
- We design a smart contract migration mechanism to reduce the ratio of CSTxs, by moving contracts often invoked simultaneously to the same shard according to the pattern learned from historical transactions.
- We implemented a prototype of LightCross based on the FISCO-BCOS project and compare its performance with two art-of-the-state sharding systems. Experimental results show that LightCross achieved 2.6× more throughput and reduced at least 50% latency.

II. Preliminaries and Related Work

In this section, we provide some necessary background and preliminaries to understand the design of LightCross.

A. Blockchain and Smart Contract

A *blockchain* is a shared and distributed ledger, which consists of a chain of *blocks*, maintained by a decentralized network of nodes. Nodes pack new transactions as blocks and append them at the end of the chain, following some consensus protocols, such as *Proof-of-Work* (PoW) [17] or *Byzantine Fault Tolerance* (PBFT) [18].

Smart contracts. A *smart contract* is a self-enforcing computer program, which executes automatically on blockchains. Ethereum [19] is the most popular blockchain platform that supports smart contracts, which are executed on the *Ethereum Virtual Machine* (EVM). Now, EVM is also integrated into other systems as the execution environment for smart contracts, such as FICOS-BCOS [20] and Ethermint [21].

States and transactions. There are two types of accounts in Ethereum, namely, the *user accounts* and the *contract accounts*, all identified by an address. A user account is associated with its Ether balance information. A contract account has an associated executable code and its own persistent storage, which is termed as the *state data* of the contract. Within each contract, the state data are represented by a set of key-value pairs that map 256-bit words to 256-bit words. The blockchain states encompass the states of all on-chain smart contracts.

A user may trigger the execution of a contract account by sending a transaction to the blockchain network, i.e., making a contract call. During the execution, the current contract state is retrieved from the blockchain, and the updated state is stored back at the end of execution. A contract can implicitly invoke a function of another contract via an internal message. There is another type of transaction, called *transfer transaction*, which merely transfers Ether between accounts without incurring any code execution on EVM.

To mine a block B, the miner must execute all transactions in B first and then broadcast B to other nodes, who append B to the end of the current chain through some consensus protocols. All other nodes execute the transactions in B serially to update their local blockchain states. After execution at each block, nodes make a snapshot of current blockchain states so that users can request each state value at a particular block height.

B. Blockchain Sharding Solutions

Sharding [4, 5, 9, 10] is an effective approach to improve the performance and scalability of existing blockchains. There are mainly three types of sharding schemes, i.e., *network sharding*, *transaction sharding*, and *state sharding*. Basically, network sharding [5, 10] divides the entire blockchain network into smaller groups, also called *shards*, the foundation of the other two sharding schemes. In transaction sharding, disjoint sets of transactions are first distributed to all shards for local consensus, and then these sets are ordered globally through a specific consensus. However, each shard still needs to synchronize and execute all transactions. The state sharding [10, 11] partitions blockchain states among different shards, i.e., a disjoint set of accounts are assigned to each shard. Sequentially, every shard only processes transactions associated with those accounts.

Limitation. Sharding is expected to enhance performance if, hopefully, most transactions are "local" to a single shard, called *intra-shard transactions*. However, the presence of *cross-shard transactions* (CSTxs), which access states from multiple shards, deeply limits the capability of sharding. Obviously, a CSTx cannot be handled by any single shard directly, thus it needs a cross-shard commit protocol to coordinate the execution for a CSTx among all involved shards. The protocols in most works [4–6, 8, 9] focus exclusively on the simplest kind of

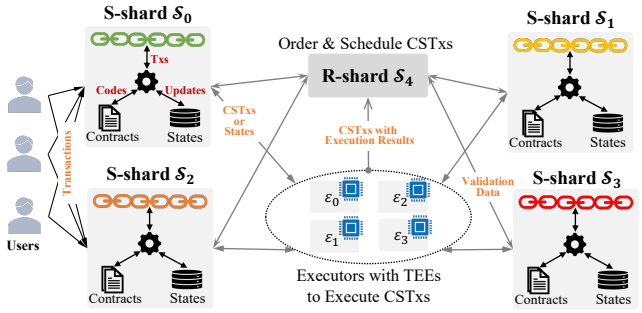


Fig. 2: Overview of LightCross architecture.

transfer CSTx, whose read/write information can be inferred statically. Thus, a transfer transaction can be split into multiple sub-transactions, each of which can be processed by a shard independently. However, obtaining state access of a contract transaction is challenging, let alone splitting transactions into irrelevant sub-transactions.

Only a few existing proposals devote to handling contract CSTxs. ByShard [12] and AHL [10] utilizes two-phase commit protocols that necessitate multiple rounds of cross-shard communication and state locking between shards to guarantee the atomicity of CSTxs. RingBFT [11] reduces the communication overhead by adhering shards to the ring order. Chainspace [15] handles contract CSTxs like the way of transfer CSTxs based on a novel contract design. Ethereum 2.0 [22] moves the contract code and state data into a centralized shard at the execution time, similar to our system. However, all these protocols incur multiple rounds of shard-to-shard communication and several rounds of consensus in involved shards.

C. TEE and Off-chain Execution

In this paper, we utilize the power of *Trusted Execution Environment* (TEE) [16] to implement a CSTx execution engine at each off-chain executor. The attestation mechanism of TEE provides auxiliary verifiable information by which others can attest to the execution results without re-executing involved contracts [23, 24]. LightCross integrates the Intel SGX as the TEE implementation. Intel SGX creates a sandbox environment called *enclave* where a contract can be executed without interference. We refer to [16] for more details about TEE.

Recently, some works propose off-chain execution [24–29] of smart contracts to empower transaction execution of blockchain. For example, Fastkitten [26] and Ekiden [25] employ TEE to implement off-chain executors to achieve high performance. However, both systems at most support single-contract or simple transfer transactions and cannot handle complex contract transactions. ACE [30] requires executors to synchronize tentative state updates of contracts from the network to execute the contracts independently. All the above solutions are designed for traditional single-ledger blockchains rather than sharding blockchains.

III. Design

A. Overview

As shown in Fig. 2, the system model of LightCross encompasses four parties: namely, *users*, *executors*, *S-shards*, and a *R-shard*.

- **Users (clients)**, are a set of clients of LightCross, who invoke transactions to perform cryptocurrency exchange or trigger state transitions of smart contracts.
- **S-shards**, are a set of shards, identified by $\{\mathcal{S}_0, \ldots, \mathcal{S}_{m-1}\}$, each of which consists of several blockchain nodes to maintain a subchain. Every smart contract is deployed at a single S-shard, which is termed as the *entry S-shard* of the contract. The entry S-shard manages the state data of the contract and processes transactions that invoke this contract.
- **Executors**, identified by $E = \{\mathcal{E}_0, \ldots, \mathcal{E}_{e-1}\}$, are nodes equipped with TEEs, designated to execute CSTxs. For every CSTx, the assigned executor requests state data of the triggered contracts from involved S-shards, executes it locally, and sends it along with read/write information as well as a generated certificate to the *R-shard* for commitment.
- **R-shard**, is a distinct shard, denoted by $\mathcal{S}_m$, to schedule the executions of CSTxs. In particular, $\mathcal{S}_m$ is responsible for sorting these CSTxs to determine a commit order after validating them. Then, $\mathcal{S}_m$ broadcasts the CSTxs to the involved S-shards for final commitment.

LightCross hides details of the sharding mechanism, e.g., locations of smart contracts, from users. A user can submit a transaction to any node of the LightCross network, and the transaction will be automatically routed to its entry S-shard of the contract invoked. [1] If a contract invoked by transaction τ does not call any other contract internally, τ is an *intra-shard transaction*. Otherwise, τ is a *cross-shard transaction*, which cannot be processed by any single S-shard. LightCross assigns a trusted executor to execute such a transaction.

Trust model. The security of LightCross depends on the security of blockchains and TEEs. Thus, we make the following assumptions about the involved technology. First, an adversary cannot break the fundamental security of blockchain. For both S-shards and R-shard, we adopt the *Byzantine Fault Tolerance* (BFT) protocol (e.g., PBFT [18]), which causes no fork, to make decisions within every shard. It requires no more than one-third nodes are malicious within a single shard in practical settings [32] to promise the safety and livness of BFT. Second, the confidentiality of contract executions is guaranteed by TEEs. An adversary cannot access the attestation private key. We note that side-channel attacks which could leak private information do exist and is a real threat to TEEs, but consider it an orthogonal issue. Mitigation exists to such attacks [25], which is outside the scope of our design. The executors can be arbitrarily malicious without breaking the oracle of TEE. For example, a malicious executor can intercept or tamper with external data to be fed into TEE or hold the messages issued by TEE forever. But an honest executor should follow our protocol correctly. We only assume there are enough honest executors to execute CSTxs, and do not require more than a certain percentage of executors to be honest, because the

[1] In this paper, we focus on smart contract transactions and the processing of non-contract CSTxs is trivial, discussed in [31].

security of LightCross does not rely on the number of honest executors.

Cross-shard transaction processing. Existing sharding systems commonly employ a heavy protocol to handle CSTxs as elaborated in Section II-B. In LightCross, we offload the CSTx execution to off-chain executors with TEE support. Then, involved S-shards can verify and accept the execution results without re-executing every CSTx. Moreover, different executors can process different CSTxs in parallel, increasing the system's parallelism. Based on the new execution scheme, we propose a lightweight cross-shard commit protocol, which avoids heavy multi-round communication between S-shards and amortizes the overhead of protocol for each CSTx by processing CSTxs in batches. This part is detailed in Section III-C.

Smart contract migration. In current sharding blockchains, every contract is assigned to a single shard randomly and permanently. However, this random-allocating approach inevitably leads to a high CSTx ratio. For example, if a transaction accesses k states in a system with m shards, the probability of having a CSTx is up to $1-(\frac{1}{m})^{k-1}$, which is approaching 1 when k or m is large. Since processing CSTxs is more costly than intra-shard transactions, the performance of sharding cannot scale as expected for this approach. Instead, in LightCross, we try to reduce the CSTx ratio by periodically redistributing the ownership of all smart contracts over all S-shards dynamically. Then, contracts that are invoked simultaneously are gathered in the same S-shard, lowering the CSTx ratio. In Section III-E, we will formally model and solve this problem.

B. Notations

The format of a transaction is defined as follows: $\tau := \langle A_s, A_r, v, \mathcal{P}, \eta\rangle$, where A_s and A_r denote the addresses of the sender and receiver of the transaction τ, respectively; v is the amount of Ether to be transferred. If A_r is a contract address, $\mathcal{P}$ includes all input arguments to the function invoked by τ. Otherwise, $\mathcal{P}$ is an empty set if A_r represents a user account. η denotes other related information, such as the nonce of A_s. Besides, each transaction piggybacks a signature issued by the sender for verification, which is neglected in this paper for brevity. We write $shards(\tau) \subseteq \{\mathcal{S}_0, \ldots, \mathcal{S}_{m-1}\}$ to denote the S-shards that are affected by τ (the S-shards contain state data that τ accesses). We say that τ is a *cross-shard transaction* if $|shards(\tau)| > 1$. Furthermore, $rshards(\tau) \subseteq shards(\tau)$ and $wshards(\tau) \subseteq shards(\tau)$ stand for the S-shards that own state data read or written by τ, respectively. Similarly, $rcontra(\tau)$ and $wcontra(\tau)$ denote all contracts read or written during the execution of τ, respectively.

We then formally define the read/write set of each transaction. The read set $R(\tau)$ of a transaction τ is defined as:

$$R(\tau) := \bigcup_{A \in rcontra(\tau)} R(\tau, A),$$

where each $R(\tau, A) = \{\langle A, \kappa, h, d\rangle\}$ is a set of states within contract A accessed by τ. For each state, κ is the key, h is the block height that indicates from which snapshot state $A[\kappa]$ is read, and d is the value of $A[\kappa]$. Similarly, the write set $W(\tau)$ of transaction τ is defined as:

$$W(\tau) := \bigcup_{A \in wcontra(\tau)} W(\tau, A),$$

where each $W(\tau, A) = \{\langle A, \kappa, d\rangle\}$ is a set of states to be written within contract A. For each state in $W(\tau, A)$, d is the new data to be stored at $A[\kappa]$.

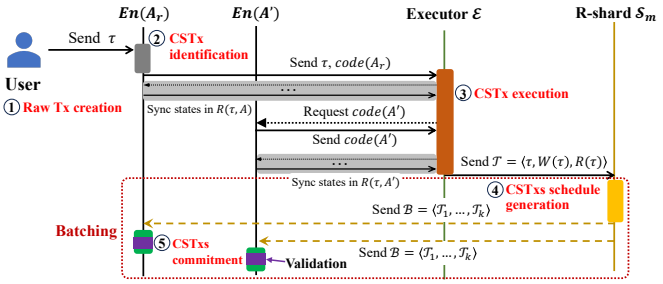


Fig. 3: Workflow of cross-shard commit protocol in LightCross.

C. Cross-Shard Commit Protocol

Every CSTx τ is assigned to an executor $\mathcal{E}$ equipped with TEE for execution. Since the memory space of TEE is limited (e.g., 128MB), $\mathcal{E}$ cannot maintain all state data within TEE. Instead, $\mathcal{E}$ stores no state data and requests state data from $shards(\tau)$ to drive the execution. This design also enables LightCross to add or remove executors as needed flexibly. The execution results generated by $\mathcal{E}$ are verifiable to others via the attestation mechanism. With this design, the intra-shard and cross-shard transactions are processed concurrently on various S-shards and executors. LightCross employs $\mathcal{S}_m$ as a coordinator to schedule all transactions to hold the *serializability*, which ensures all S-shards operate like a single-ledger blockchain that executes transactions serially. The detailed cross-shard commit protocol goes through five phases as follows (also see Fig. 3).

① **Raw transaction τ creation.** The sender account A_s first creates a raw transaction $\tau = \langle A_s, A_r, v, \eta, \mathcal{P}\rangle$ that invokes the smart contract with address A_r. Then, the transaction τ is routed to the entry S-shard, denoted by $En(A_r)$, of the contract A_r.

② **CSTx identification.** Once τ in $En(A_r)$ is packed in the next block, it will be executed speculatively by running the code associated with A_r. If the execution of τ calls another contract A' on another S-shard $En(A')$, τ is marked as a CSTx.[2] Sequentially, $En(A_r)$ forwards τ to an executor for execution alongside the code $code(A_r)$ of A_r.

③ **CSTx execution.** Once receiving a CSTx τ, an executor $\mathcal{E}$ first initializes a new *execution engine* implemented with the TEE. $\mathcal{E}$ starts the execution by loading $code(A_r)$. During the execution, $\mathcal{E}$ requests values of states in the read set $R(\tau, A_r)$ from $En(A_r)$ through a point-to-point communication, which we will explain later. When another contract A' is invoked

[2]Alternatively, we can do such identification through static program analysis techniques [33] without executing transactions.

internally, $\mathcal{E}$ requests the code $code(A')$ from the counterpart entry S-shard $En(A')$ to drive execution and then requests values of states in $R(\tau, A')$ in the same way. After execution, $\mathcal{E}$ generates a certificate $Cert_\tau$ for attestation and sends a message, $\mathcal{T}_i = \langle \tau, W(\tau), R(\tau), Cert_\tau \rangle$ to $\mathcal{S}_m$, including the original transaction and its read/write set.

④ **CSTxs schedule generation.** The R-shard $\mathcal{S}_m$ schedules a batch of received messages $\{\mathcal{T}_i\}$ periodically. First, upon receiving a message $\mathcal{T}_i$, $\mathcal{S}_m$ verifies the execution by attesting $Cert_\tau$ to ensure $W(\tau)$ and $R(\tau)$ are correct. Then, $\mathcal{S}_m$ packs a batch of valid messages $\mathcal{B} = \langle \mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k \rangle$ for consensus and sends $\mathcal{B}$ to all the S-shards in $\bigcup_{\tau \in \mathcal{B}} shard(\tau)$ after consensus.

⑤ **CSTxs commitment.** When receiving $\mathcal{B}$, each S-shard $\mathcal{S}_i \in \bigcup_{\tau \in \mathcal{B}} rshard(\tau)$, which is going to be read from, collaborates with $\mathcal{S}_m$ to validate every transaction in $\mathcal{B}$ and aborts those that violate the serializability (elaborated in Section III-D). We write $P = \langle \mathcal{T}_{q_1}, \mathcal{T}_{q_2}, \ldots, \mathcal{T}_{q_s} \rangle$ $(1 \leq q_1 < \ldots < q_s \leq k)$ to denote the remaining valid transactions. Then each $\mathcal{S}_i$ further extracts all CSTxs, denoted by $\langle \mathcal{T}_{p_1}, \ldots, \mathcal{T}_{p_t} \rangle$ $(\{p_1, \ldots, p_t\} \subseteq \{q_1, \ldots, q_s\})$, that $\mathcal{S}_i$ is involved with, such that $\mathcal{S}_i \in shard(\mathcal{T}_{p_j}.\tau)$ $(1 \leq j \leq t)$. Finally, $\mathcal{S}_i$ inserts $\langle \mathcal{T}_{p_1}, \ldots, \mathcal{T}_{p_t} \rangle$ into the next block and commits them upon agreement, by applying the writes declared in write sets of these transactions.

Communication primitive. In the above protocol, LightCross needs a Byzantine-resilient primitive that enables coordination between different parties, i.e., the R-shard, S-shards, and executors. In a potentially hostile environment, the communication mechanism should be secure and reliable to defend against adversarial attacks. For example, when a correct executor requests state data from an S-shard, the communication mechanism should promise that the correct executor can eventually receive the correct state data even though malicious nodes exist. LightCross leverages the built-in Merkle trees of blockchains to validate the state data delivered from S-shards to executors, which is a point-to-point verifiable sending (see Appendix D in [31]). With such approach, an executor only has to access a few nodes in an S-shard (<3 for most cases) to obtain requested data values. Besides, we may choose any cluster-sending protocols [34–36] to enable communication between S-shards and the R-shard.

Advantages. The advantages of our protocol over current approaches come from the two aspects. First, the execution for a CSTx is centralized to a single executor. Thus, there is no need to split the CSTx, and the logic of contracts invoked can be arbitrarily complex. In contrast, existing approaches require multi-round shard-to-shard communication, which is costly and time-consuming, between the involved shards, to process a CSTx. In LightCross, executors adopt a point-to-point mechanism to synchronize state data, which saves a great number of data to be transferred. Second, LightCross amortizes the costs for a group of CSTxs by batching, as demonstrated in the dotted rectangle in Fig. 3. Specifically, CSTxs are concurrently executed at executors in an optimistic way and then R-shard finds a feasible schedule for them in a batch. Conversely, existing approaches commonly have to initiate such a protocol instance for every single CSTx, raising the system burden.

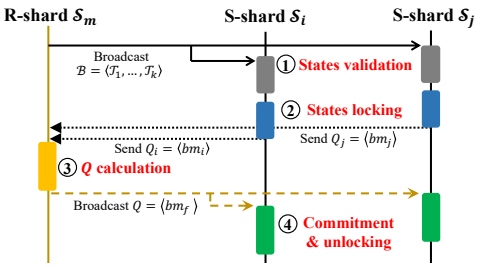


Fig. 4: Workflow of transaction validation.

The protocol presented so far does not promise the correctness of the concurrent CSTx execution. For instance, a CSTx in $\mathcal{T}_{p_i}$ may read the value d of $A[\kappa]$ at a block height h. However, before $\mathcal{T}_{p_i}$ is committed by $En(A)$, $A[\kappa]$ may be updated to d' by another intra-shard transaction. Subsequently, $\mathcal{T}_{p_i}$ will miss the latest value and cannot be serialized. To address this issue, LightCross adds a validation process into the last phase, which we present next.

D. Transaction Validation

To prevent CSTxs that have read stale state values from being committed, R-shard $\mathcal{S}_m$ validates if a state is updated after being read. If so, $\mathcal{S}_m$ discards these transactions. The challenge lies in that the validation should be lightweight enough, especially incurring less communication overhead between S-shards and R-shard. To this end, shards in LightCross exchange information based on concise bitmap structures, and the validation protocol goes through the following steps (also see Fig. 4):

① Once $\mathcal{B} = \langle \mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k \rangle$ gets agreed in R-shard, $\mathcal{S}_m$ broadcasts $\mathcal{B}$ to all S-shards in $\bigcup_{\tau \in \mathcal{B}} shard(\tau)$. Each S-shard $\mathcal{S}_i \in rshard(\tau)$ checks the read set $R(\tau)$ of every transaction τ in $\mathcal{B}$. For each state $\langle A, \kappa, h, d \rangle$ in $R(\tau)$, $\mathcal{S}_i$ checks whether the value d of $A[\kappa]$ at the block height h is identical to its latest value. If not, τ is invalid due to a stale read on $A[\kappa]$.

② S-shard $\mathcal{S}_i$ sends a message $\langle bm_i \rangle$ to the R-shard $\mathcal{S}_m$, where bm_i is a bitmap indicating whether each transaction is valid or not. Meanwhile, $\mathcal{S}_i$ locks all states in the read sets of the valid transactions in $\mathcal{B}$. As a result, any other transactions, trying to write these states, will not be processed before the locked states are released.

③ Once $\mathcal{S}_m$ collects all messages $\{\langle bm_i \rangle\}$ from S-shards in $\bigcup_{\tau \in \mathcal{B}} rshard(\tau)$, $\mathcal{S}_m$ merges all bitmaps to get the final bitmap $bm_f = \bigwedge_i bm_i$, which indicates every transaction is valid or not at the final stage. Then, $\mathcal{S}_m$ broadcasts $\langle bm_f \rangle$ to S-shards in $\bigcup_{\tau \in \mathcal{B}} shard(\tau)$.

④ Upon receiving $\langle bm_f \rangle$ from $\mathcal{S}_m$, each S-shard $\mathcal{S}_i$ extracts transactions $\langle \mathcal{T}_{p_1}, \ldots, \mathcal{T}_{p_t} \rangle$ out of $\langle \mathcal{T}_{q_1}, \ldots, \mathcal{T}_{q_s} \rangle$, the final valid transactions, and commits them as stated before. Finally, $\mathcal{S}_i$ releases all locks on states.

Apart from maintaining the validity of a single CSTx, the R-shard should also eliminate conflicts between CSTxs in $\mathcal{B}$. In particular, if a CSTx $\mathcal{T}$ reads a state $A[\kappa]$, while another

CSTx $\mathcal{T}'$ writes $A[\kappa]$, they conflict with each other. A naïve solution is that $\mathcal{S}_m$ orders transactions corresponding to their submitted timestamps and discards any transaction that has read stale values. Some previous works [37] are also proposed to derive an optimal precisely scheduling of transactions to reduce the abort ratio. For instance, if $\mathcal{T}$ is scheduled ahead of $\mathcal{T}'$, both of them can be executed correctly without causing any abort. Any of these solutions would work and it is not the focus of this paper.

Notably, as a part of the protocol, some states are locked from step ② to ④, potentially blocking other transactions' state write. In practice, this lock period is insignificant. First, the bitmaps exchanged between shards only require a tiny amount of data transmission. Second, when each S-shard achieves consensus on CSTxs, it only performs verification with no need of executing them. Third, the batching strategy amortizes the overhead for each CSTx. Therefore, the lock mechanism in LightCross is lightweight enough to incur no significant delay. Besides, although LightCross merely employs one single R-shard to coordinate the schedule, it will not be the system's bottleneck. This is because R-shard just orders and verifies CSTxs beyond executing them, and the throughput of current PBFT (about 8K tps) on R-shard can serve tens of S-shards. The experimental results in Section V also confirm this view.

E. Smart Contract Migration

In most sharding blockchains, smart contracts are distributed over all shards randomly and permanently, leading to a high CSTx ratio. Although our cross-shard commit protocol is lightweight, it cannot compensate for the performance degradation due to a high CSTx ratio. Thus, some works, e.g., BrokerChain [8] and OptChain [2], devote to lowering the CSTx ratio. However, these solutions just work for transfer transactions. In LightCross, we try to reduce the CSTx ratio towards smart contract transactions. Basically, LightCross dynamically migrates contracts, often triggered within the same transactions, to the same S-shard. Under such conditions, many CSTxs can be avoided and reduced to much cheaper intra-shard transactions.

First of all, we need to model this problem formally. In LightCross, we model the relationship between smart contracts with a *transaction call graph* (Definition 1) and then convert the smart contract migration into an offline graph partition problem with temporal balancing.

Definition 1 (Transaction call graph)**.** *A transaction call graph (TCG) is a graph $G = \langle V, E\rangle$, where each vertex in V represents a pair $\langle A, w\rangle$ of contract A and its weight w, and each edge $\langle A_1, A_2, cnt\rangle \in E$ connects two contracts A_1 and A_2. Specifically, w is the number of transactions that trigger A (including implicit calls) and cnt counts the number of transactions that invoke A_1 and A_2 simultaneously.*

We build this TCG from historical transactions and partition it with two goals. (1) **balanced workload**: partition the vertices in G into equal disjoint subgraphs approximately; and (2) **low CSTx ratio**: minimize the sum of cnts of *crossing edges*, i.e., an edge whose vertices belong to two different subgraph. Then, the vertices in a subgraph consist of a subset of contracts and each subset should be assigned to a S-shard. Intuitively, the edge in TCG that connects two contracts often invoked simultaneously will be labeled with a greater weight. Thus, the partition algorithm will prefer to put the two contracts into the same subset, transforming the transactions that call both contracts to intra-shard ones.

Formally, given a TCG G and m S-shards, LightCross partitions G into m disjoint subgraphs $\{G_0, \ldots, G_{m-1}\}$, where $G_i = \langle V_i, E_i\rangle$ and for any $i \neq j$, $V_i \cap V_j = \emptyset$. More importantly, respecting the aforementioned goals, the partition optimizes the two objectives. **1)** The weight $\mathcal{W}(G_i)$ of each subgraph should be close to $\mathcal{W}(G)/m$ as much as possible, where $\mathcal{W}(G_i)$ is defined as $\mathcal{W}(G_i) = \sum_{\langle A,w\rangle \in V_i} w$; and **2)** we should minimize following function:

$$\min \sum_{0 \leq i < j < m} EdgeCnt(G_i, G_j). \tag{1}$$

In Eq. (1), $EdgeCnt(G_i, G_j)$ sums up the cnts of cross edges connecting G_i and G_j. This multi-objective optimization problem in graph mining can be solved by Metis [38].

Given a partition, we should consider how to assign subgraphs to all S-shards to minimize the migrated data. As stated above, all contracts in a subgraph G_i are assigned to a single S-shard, denoted by $\mathcal{S}_{\sigma(i)}$, by migrating the state data of these contracts to $\mathcal{S}_{\sigma(i)}$. Here, $\sigma(\cdot)$ is a permutation that maps $\{0, \ldots, m-1\}$ to $\{0, \ldots, m-1\}$. Let $D_{i,j}$ be the size of data to be migrated from $\mathcal{S}_{\sigma(i)}$ to $\mathcal{S}_{\sigma(j)}$ against a partition. Then we pick the optimal permutation $\hat{\sigma}(\cdot)$ that minimizes $\sum_{1 \leq i \neq j < m} D_{i,j}$, the total state data to be moved.

Since the overhead of smart contract migration remains high, LightCross uses three additional strategies to relieve this issue. First, we introduce continuous epochs $\{e_1, e_2, \ldots\}$ and the migration only happens at the beginning of each epoch. The chosen epoch duration should be long enough to amortize the migration overhead. Second, we try to reduce contract migration and avoid moving large-size contracts. To this end, we assign a higher weight to edges connecting intra-shard contracts. We replace the edge weight in Eq. (1) with $cnt' = \lambda \cdot Sigmoid(d_1 + d_2) \cdot cnt$, where λ is a coefficient, $Sigmoid(\cdot)$ is the *sigmoid function*, and d_1, d_2 are the sizes of the state data of A_1 and A_2, respectively. The coefficient λ is set to a number >1 if A_1 and A_2 belong to the same S-shard, and <1 otherwise. If the CSTx ratio of the previous epoch is under a reasonable threshold r_c, LightCross skips the migration procedure for the upcoming epoch since the current contract distribution is considered satisfactory.

Workflow. When a new epoch e_k starts, R-shard $\mathcal{S}_m$ is in charge of coordinating the contract migration process. Figure 5 shows an example of transaction call graph construction, graph partition, and smart contract migration with four S-shards. The detailed workflow is described as follows.

① **Preparation.** At the end of epoch e_{k-1}, $\mathcal{S}_m$ stops scheduling any CSTx and notifies all S-shards to prepare for the

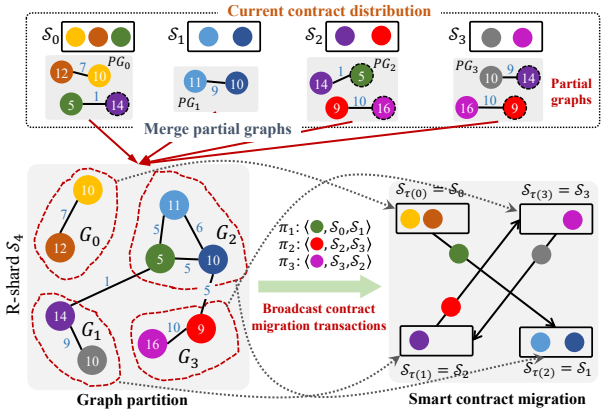


Fig. 5: Example of smart contract migration.

upcoming new epoch. Then each S-shard completes remaining transactions and sends a confirming message to $\mathcal{S}_m$.

② **TCG construction.** To construct the TCG for e_k, every S-shard $\mathcal{S}_i$ collects transactions that happened in e_{k-1} to construct a partial graph PG_i as shown at the top of Fig. 5 (the dotted cycles are contracts not in current S-shards). Then PG_i is sent to R-shard $\mathcal{S}_m$. Once receiving all partial graphs $\langle PG_0, \ldots, PG_{m-1} \rangle$, $\mathcal{S}_m$ merges them into a complete TCG G. We can compress the TCG by ignoring contracts that are rarely in, and the details can be found in Appendix B of [31].

③ **Graph partition.** $\mathcal{S}_m$ leverages the Metis algorithms [38] to partition graph G while achieving the objectives mentioned above. Then $\mathcal{S}_m$ decides the permutation $\hat{\sigma}(\cdot)$, as well as the contracts to be migrated. For every contract A, $\mathcal{S}_m$ creates a special type of transaction $\pi_i = \langle A, \mathcal{S}_s, \mathcal{S}_t \rangle$, requesting that the ownership of contract A to be transferred from $\mathcal{S}_s$ to $\mathcal{S}_t$. Such transactions $\{\pi_i\}$ are broadcast to involved S-shards after being agreed by the R-shard.

④ **State data migration.** Upon receiving every contract migration transaction $\pi_i = \langle A, \mathcal{S}_s, \mathcal{S}_t \rangle$, the S-shard $\mathcal{S}_s$ begins to move the state data of the target contract A to S-shard $\mathcal{S}_t$.

⑤ **Service resuming.** After completing contract migration transactions, each S-shard sends a message to R-shard, indicating it can restart processing transactions. R-shard will schedule CSTxs involved in resumed S-shards.

The state data of every contract is stored as a Merkle subtree in LightCross like Ethereum. However, moving all structures directly may be time-consuming and suspend the service for a long. Instead, LightCross could move contract state data asynchronously as proposed in [36], where every S-shard can process transactions while contracts are being transmitted. Besides, since the ownership of contracts varies over time, LightCross maintains a dynamic distributed routing table to record this information to navigate nodes to route transactions to their entry S-shards. This part is detailed in [31].

IV. Discussion and Implementation

In this section, we discuss the correctness of LightCross and present its implementation details. Due to space limit, proofs of related theorems are presented in Appendix E of [31].

A. System Correctness Analysis

Atomicity. Essentially, the cross-shard commit protocol has to promise the atomicity, meaning that *all S-shards involved in a CSTx should either commit, or abort without making changes to the state*. Theorem 1 ensures the atomicity of CSTx in LightCross.

Theorem 1. *If an S-shard in $shards(\tau)$ commits a CSTx τ, all the other S-shards in $shards(\tau)$ must commit τ eventually.*

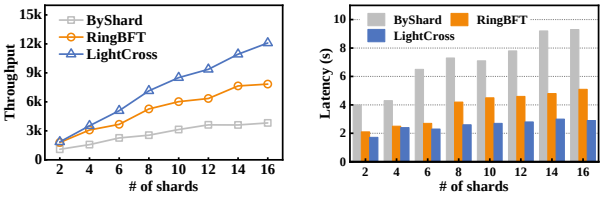
Serializability. The presence of CSTxs makes the processing of transactions interleaved in different S-shards, i.e., a CSTx is processed on multiple S-shards concurrently. Therefore, the cross-shard commit protocol should promise that the results of these concurrent executions of CSTxs are correct. Particularly, at any time point, *the results generated by committed transactions at all S-shards should be equivalent to that of a serial execution in some order*, which is termed as the *serializability*. This ensures all S-shards in LightCross operate like a single-ledger blockchain, which executes transactions serially. Theorem 2 promises our protocol is serializable.

Theorem 2. *If an S-shard commits a transaction τ (intra-shard or cross-shard), τ will be serialized properly and eventually.*

Liveness. The liveness of LightCross encompasses two aspects. First, each S-shard can keep generating blocks and will never be blocked at a particular block height. This is guaranteed by the liveness of PBFT consensus protocol [18, 39]. PBFT ensures that the consensus at each block height will terminate within a finite period, so each S-shard can continuously produce blocks. Second, every contract deployed on-chain is responsive, i.e., a user invoking the contract can eventually receive executing results regardless of success or abort. If a transaction is intra-shard, it is routed to its entry S-shard for execution, and the liveness of PBFT promises it will be executed eventually. For a CSTx τ, LightCross ensures that it will be correctly executed by an executor $\mathcal{E}$ and then sent to the R-shard for commitment. To cope with the failure of $\mathcal{E}$, LightCross sets a timeout parameter t_{expire}. If the entry S-shard $En(\tau.A_r)$ cannot commit or abort τ in time, $En(\tau.A_r)$ will reassign τ to another $\lambda+1$ executors, where λ denotes the times of time expiration. This gets more executors executing τ after every expiration, raising the success probability.

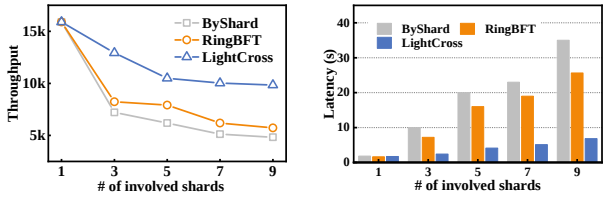
B. Implementation

Design of S-shard and R-shard. S-shards are responsible for managing smart contracts, performing consensus on blocks, and executing transactions. We implemented the S-shard prototype based on FISCO-BCOS [20], a permissioned blockchain system with the support of EVM and built-in PBFT consensus. In each S-shard, the state data of contracts are organized as a Merkle tree, and the root hash is also included in the block headers as Ethereum does. The only difference is that each S-shard only manages a subset of all contracts. For R-shard, we leverage PBFT consensus to order CSTxs and integrate the Metis tool to partition transaction call graphs. Besides, R-shard adopts a



(a) Throughput (b) Latency

Fig. 6: Impact of shards (S-shards in LightCross).



(a) Throughput (b) Latency

Fig. 7: Impact of involved shards (S-shards in LightCross).

cluster-sending protocol [36] to exchange messages for cross-shard commit protocol with S-shards.

Design of Executors. The executors in LightCross are implemented with Intel SGX [40], which provides isolated hardware enclaves and a fundamental Software Development Kit to realize the functions. To support contract execution in these enclaves, we employ Occlum [41], a lightweight library OS for Intel SGX, to convert non-TEE programs to TEE programs. On every executor, an execution engine consists of three enclaves: 1) a *data enclave* for data transmission between the internal enclaves and the external environments; 2) a *key enclave* for key management; and 3) an *execution enclave* for contract execution. These enclaves communicate securely via the local attestation of the Intel SGX, which guarantees mutual trust for enclaves on the same platform. During execution, the data enclave synchronizes necessary state data from S-shards through a verifiable send. For a CSTx τ, the execution enclave generates a certificate $Cert_\tau$, which can be verified via remote attestation.

V. Experimental Evaluation

A. Setup

Comparisons. In all experiments, we compared the performance of LightCross against two other state-of-the-art sharding techniques supported by the BFT protocols, namely, RingBFT [11] and ByShard [12]. For a fair comparison, we also implemented both systems onto the FISCO-BCOS project.

For ByShard, a reference committee is responsible for dispatching a CSTx to all involved shards, and each shard executes a code fragment while locking the states to be accessed, i.e., similar to a sub-transaction that only reads or writes local state data, independently. However, complex smart contract transactions cannot be easily split into such independent sub-transactions. Therefore, to apply ByShard to more complex transactions in our experiment, we manually declare the read set of states and the contracts to be invoked in each transaction. During the protocol execution, once the states to be read are locked, and all involved shards share these state values, so that all of them can execute the entire CSTx independently.

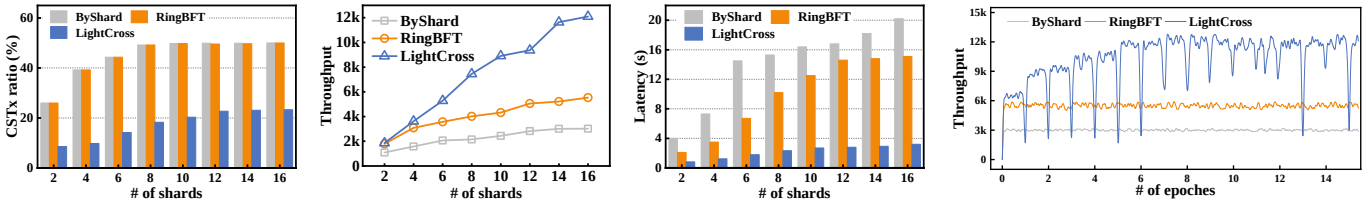
Datasets. To test the performance of different sharding systems on real-world workloads, we first synchronized the transactions from the mainnet of Ethereum during the period between Jan 1, 2022 and April 30, 2022 (769,020 blocks in total). There are 122 million transactions in total, and about 84 million of them (69%) made contract calls, of which more than 40% transactions invoke at least two smart contracts. To enable large-scale experiments, we analyzed the real-world workloads to build a synthetic workload generator to generate enough transactions.

Testbed. We evaluated the performance of LightCross on Ubuntu 20.04.1. The experimental machine has 20 Intel(R) Xeon(R) Silver 4210 CPUs @ 2.20GHz, 96GB memory, and a 4TB disk. Intel SGX SDK v2.2 and Occlum v0.28.0 were used at the time of the experiment. We deployed up to 16 S-shards and each S-shard has 22 nodes. We ran multiple (up to 12) instances of S-shard nodes on each physical machine to simulate more virtual nodes. Since all machines are located at the same data center, we divided all S-shards nodes into 6 groups, simulating 6 globally distributed regions. Then we limited the bandwidth between nodes from different regions to simulate a real network environment (refer to Table 1 in [35]). Besides, we deployed sufficient executors in every group to avoid executions being the performance bottleneck.

B. Evaluation Results

Scaling number of S-shards. First, we studied the effect of scaling the number of shards (or S-shards in LightCross). Specifically, we selected CSTxs that can access from 2-16 shards and always ensure 30% of smart contract transactions are CSTxs. We tested the throughput and latency of intra-shard and cross-shard transactions with 2-16 shards in every system. We used Fig. 6 to illustrate the throughput and latency metrics. LightCross achieves 4× and 1.6× higher throughput than ByShard and RingBFT in the 16 S-shard setting, respectively. As the number of shards rises, the number of involved shards per CSTx may increase accordingly. Then an executor needs to communicate with more S-shards to synchronize state data, and in other approaches, more shards engage in the execution of a CSTx. However, shard-to-shard communication's overhead is higher than point-to-point communication in LightCross. From 3 to 16 shards, the latency of LightCross increases from 1.65s to 2.4s since the overhead of point-to-point communication is relatively low. In the case of ByShard, the overhead of state synchronization increases quickly due to the heavy shard-to-shard communication. Moreover, the protocols for CSTxs are all coordinated by a special reference committee. RingBFT scales better than ByShard, since no reference committee is needed to lead the cross-shard commit protocols. However, we still observe a drop in the throughput of RingBFT, because the number of communication rounds between shards is linearly correlated with the number of involved shards.

Varying number of involved shards. Second, we now keep the number of shards fixed at 16 and select transactions



(a) CSTx ratio (b) Throughput (c) Latency (d) Throughput over time

Fig. 8: Impact of smart contract migration.

that access 1-9 shards (or S-shards). Figure 7 illustrates the throughput and latency metrics. As expected, all three approaches observe a drop in performance as the number of involved shards increases. Still, LightCross outperforms the other two approaches. Furthermore, with more shards involved per CSTx, the performance gap between LightCross and the other two approaches enlarges. When the number of involved shards is 1, all transactions access the same shard. In this case, all three approaches achieve the same throughput and latency as there is no CSTx, and the capability of executing intra-shard transactions is similar for all approaches. Particularly, they achieve 15.8K transactions per second throughput among 16 globally distributed shards. With more shards involved, LightCross leads the throughput race over the other two approaches with an increasing margin: from 4% to 200% (see Fig. 7(a)). In particular, when the number of involved shards is greater than 3, the latency of CSTx processing for ByShard and RingBFT is increased significantly. Since both approaches need to lock the involved states during the entire process, many intra-shard transactions, which attempt to access these locked states, are also blocked. Another major reason for the extra latency is the increased communication overhead caused by CSTxs, which has been analyzed in previous tests.

Impact of smart contract migration. Third, we studied the impact of the effectiveness and efficiency of our smart contract migration. In these tests, we fixed the number of shards at 16 and made 50% transactions access more than one smart contracts. In LightCross, we configured the length of an epoch to about 6 hours. For all approaches, every single smart contract is assigned to a shard according to the first few bits of its address. Figure 8 demonstrates the performance of all three approaches against varying numbers of shards. In Fig. 8(a), as the number of shards in systems increases, the CSTx ratio of all approaches grows accordingly. Since ByShard and RingBFT cannot change the ownership of contracts, the CSTx ratio quickly reaches its peak (about 50%). Instead, the CSTx ratio in LightCross still remains 25% with 16 S-shards. Figure 8(b) presents the throughput of three approaches against varying numbers of shards. As expected, the improvement for ByShard and RingBFT is not prominent. For example, the throughput of ByShard and RingBFT has just increased about 3× when the number of shards increase from 3 to 16. In contrast, LightCross achieves a 6.4× speedup because LightCross reduces the CSTx ratio via contract migration.

Figure 8(c) reports the latency of transactions for three approaches. As explained above, with more shards, the CSTx ratio also increases. However, the cross-shard commit protocols in ByShard and RingBFT are heavy-weight, which incur multi-round shard-to-shard communication. Therefore, the latency of transactions in ByShard and RingBFT reaches 21s and 15s, respectively, when there are 16 shards. Due to the lightweight cross-shard commit and low CSTx ratio, the latency of LightCross remains low (up to 3.2s). Fig. 8(d) demonstrates the throughput of three approaches over time. At the beginning, ByShard and LightCross perform similarly, because the initial contract assignment leads to a high CSTx ratio. Notably, the CSTx ratio for LightCross reduces gradually over time due to periodical contract migrations, resulting in improved throughput (see Fig. 8(d)). Around the end of every epoch (or start of the next one), there is a big drop in throughput due to smart contract migration, which finishes in 10–30 minutes. Benefiting from lazy synchronization, each S-shard can quickly resume processing transactions during this period. Even though LightCross does not perform such migration during some epochs, the throughput is still relatively low. This is because LightCross needs to notify all S-shards about the decision while others should wait. After about six epochs, the contract distribution in LightCross becomes stable, and the smart contract migration is seldom triggered. In contrast, the throughput of ByShard and RingBFT always stays low because the ownership of contracts are static.

VI. Conclusion

We present LightCross, a sharding blockchain, to process arbitrarily complex cross-shard smart contract transactions efficiently. At its core, LightCross offloads the execution for CSTxs from S-shards to a set of off-chain TEE-empowered executors, without incurring multiple rounds of cross-shard communication. To commit the CSTxs, LightCross employs a lightweight cross-shard commit protocol with guarantees of atomicity, serializability, and liveness. Furthermore, LightCross integrates a dynamic smart contract migration mechanism to reduce the CSTx ratio, avoiding a great number of CSTxs and increasing the overall performance significantly. Finally, we implement LightCross and the evaluation results demonstrate the superiority of our system.

Acknowledgments

This work was supported by the Nanyang Technological University Centre for Computational Technologies in Finance (NTU-CCTF). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NTU-CCTF.

REFERENCES

[1] Y. Tao, B. Li, J. Jiang, H. C. Ng, C. Wang, and B. Li, "On sharding open blockchains with smart contracts," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1357–1368.

[2] L. N. Nguyen, T. D. Nguyen, T. N. Dinh, and M. T. Thai, "Optchain: optimal transactions placement for scalable blockchain sharding," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 525–535.

[3] H. Huang, Z. Huang, X. Peng, Z. Zheng, and S. Guo, "Mvcom: Scheduling most valuable committees for the large-scale sharded blockchain," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2021, pp. 629–639.

[4] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 583–598.

[5] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 17–30.

[6] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," in *16th USENIX symposium on networked systems design and implementation (NSDI 19)*, 2019, pp. 95–112.

[7] M. Zhang, J. Li, Z. Chen, H. Chen, and X. Deng, "Cycledger: A scalable and secure parallel protocol for distributed ledger via sharding," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 358–367.

[8] H. Huang, X. Peng, J. Zhan, S. Zhang, Y. Lin, Z. Zheng, and S. Guo, "Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding," in *IEEE INFOCOM*, 2022.

[9] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 931–948.

[10] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *Proceedings of the 2019 international conference on management of data*, 2019, pp. 123–140.

[11] S. Rahnama, S. Gupta, R. Sogani, D. Krishnan, and M. Sadoghi, "Ringbft: Resilient consensus over sharded ring topology," *arXiv preprint arXiv:2107.13047*, 2021.

[12] J. Hellings and M. Sadoghi, "Byshard: Sharding in a byzantine environment," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2230–2243, 2021.

[13] Z. Hong, S. Guo, P. Li, and W. Chen, "Pyramid: A layered sharding blockchain system," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[14] M. Li, Y. Lin, J. Zhang, and W. Wang, "Jenga: Orchestrating smart contracts in sharding-based blockchain for efficient processing," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2022, pp. 133–143.

[15] A. Sonnino, "Chainspace: A sharded smart contract platform," in *Network and Distributed System Security Symposium 2018 (NDSS 2018)*, 2018.

[16] V. Costan and S. Devadas, "Intel sgx explained," *Cryptology ePrint Archive*, 2016.

[17] S. Nakamoto *et al.*, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[18] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[19] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[20] "FISCO-BCOS," http://fisco-bcos.org/, 2023.

[21] I. S. Cardenas, J. B. May, and J.-H. Kim, "Automatadao: A blockchain-based data marketplace for interactive robot and iot data exchanges using ethermint and state channels," *Blockchain Technology for IoT Applications*, pp. 17–38, 2021.

[22] "Ethereum sharding," https://ethereum.org/en/roadmap/danksharding/, 2023.

[23] Y. Yan, C. Wei, X. Guo, X. Lu, X. Zheng, Q. Liu, C. Zhou, X. Song, B. Zhao, H. Zhang *et al.*, "Confidentiality support over financial grade consortium blockchain," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2227–2240.

[24] C. Xu, C. Zhang, J. Xu, and J. Pei, "Slimchain: scaling blockchain transactions through off-chain storage and parallel processing," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2314–2326, 2021.

[25] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 185–200.

[26] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi, "Fastkitten: Practical smart contracts on bitcoin," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 801–818.

[27] J. Liu, P. Li, R. Cheng, N. Asokan, and D. Song, "Parallel and asynchronous smart contract execution," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 5, pp. 1097–1108, 2021.

[28] K. Wüst, L. Diana, K. Kostiainen, G. Karame, S. Matetic, and S. Capkun, "Bitcontracts: Supporting smart contracts in legacy blockchains," *Cryptology ePrint Archive*, 2019.

[29] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable, private smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1353–1370.

[30] K. Wüst, S. Matetic, S. Egli, K. Kostiainen, and S. Capkun, "Ace: Asynchronous and concurrent execution of complex smart contracts," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 587–600.

[31] "LightCross Appendix," https://sites.google.com/view/infocom24, 2023.

[32] D. Dolev, "Unanimity in an unknown and unreliable environment," in *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*. IEEE, 1981, pp. 159–168.

[33] "Slither tool," https://github.com/crytic/slither, 2022.

[34] J. Hellings and M. Sadoghi, "Brief announcement: The fault-tolerant cluster-sending problem," in *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[35] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, "Resilientdb: Global scale resilient blockchain fabric," *arXiv preprint arXiv:2002.00160*, 2020.

[36] X. Qi, "S-store: A scalable data store towards permissioned blockchain sharding," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022, pp. 1978–1987.

[37] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, and B. C. Ooi, "A transactional perspective on execute-order-validate blockchains," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 543–557.

[38] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, 1996, pp. 35–es.

[39] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, University of Guelph, 2016.

[40] "Intel," https://software.intel.com/sites/default/files/managed/1b/a2/Intel-SGX-Platform-Services.pdf, 2023.

[41] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of intel sgx," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 955–970.

[42] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *Peer-to-Peer Systems: First InternationalWorkshop, IPTPS 2002 Cambridge, MA, USA, March 7–8, 2002 Revised Papers*. Springer, 2002, pp. 53–65.

[43] R. C. Merkle, "A certified digital signature," in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 218–238.

APPENDIX

A. Account Segmentation

The accounting system of LightCross is the same as in Ethereum, including two types of accounts: *user account*s (UAs) and *contract account*s (CAs). A private key controls a UA, has no associated code and can issue transactions. A contract account has an associated executable code. In this paper, LightCross distributes each contract to a distinct S-shard while dealing with UAs in another way. Commonly, there are far more UAs than CAs in a blockchain, so many CSTxs will arise if we distribute UAs over all S-shards. Since UAs have no executable codes and persistent state data, a transaction can transfer tokens to a UA or deduct tokens from it. Fortunately, tokens are fungible, so that LightCross segments an account into multiple sub-accounts and distributes them over all S-shards, recognized as the *account segmentation mechanism* [8].

LightCross opens a sub-account a_i for an user account a in the S-shard $\mathcal{S}_i$ automatically if needed, which is transparent to users. If a transaction τ transfers v tokens from smart account A to another user account a, which is not in $\mathcal{S}_i = En(\tau.A)$ currently, $\mathcal{S}_i$ then creates the sub-account a_i with v tokens initially. With a sub-account a_i, other transaction that tries to transfer v tokens from a in $\mathcal{S}_i$ can directly deduct v from a_i if its balance $a_i.bal$ is sufficient.

Segments accumulation. If the balance $a_i.bal$ is insufficient in a transaction τ or τ has to gain the total balance of a, LightCross will accumulate all sub-accounts into a complete account, which can be termed as a cross-shard transaction as well. In this case, LightCross processes τ through the cross-shard commit protocol by gathering the balance of all sub-accounts in the executor. Notably, LightCross will write back the balance of a to an optimal S-shard $\mathcal{S}_i$, at which $a_i.bal$ is the biggest. Besides, if τ has gathered enough balance to pay, the executor will stop collecting balance from other sub-accounts.

B. TCG compression

As more smart contracts are deployed on-chain, the number of vertexes and edges in TCG grows accordingly, raising the overhead of partial TCG transmission. We can compress this TCG by omitting contracts seldom invoked. After investigating the existing smart contracts in Ethereum, we observe that a great part of smart contracts is invoked so few times. When forming TCG, each S-shard counts these contracts whose weight w is over a threshold $Thres$. Then, the contracts rarely involved in a few transactions are deducted from TCG, significantly reducing the size of TCG. This approach will not affect the effectiveness of contract migration because the contracts, which are frequently invoked, attribute most to the overhead of contract migration and CSTxs processing.

C. Transaction Routing

In LightCross, the sharding architecture is transparent to users. Thus a user does not need to know the entry S-shard of every contract, requiring LightCross should automatically route submitted transactions to counterpart S-shards for further processing. To this end, LightCross maintains a routing table RT to record the entry S-shard of every contract. Specifically, the routing table consists of tuples, each in the form $\langle A, \gamma \rangle$. Here, A is the contract's address, and γ, the distinct index of an S-shard, such that $S_\gamma = En(A)$.

We can require every S-shard reserves a full copy of RT. However, as more contracts are deployed, it adds extra storage overhead to each S-shard. LightCross, alternatively, partitions the RT into m sub-tables and assigns each sub-table to a distinct S-shard. A realistic partition is to use the modulo approach

$$\gamma' = A \bmod m$$

to determine the S-shard that stores tuple $\langle A, \gamma \rangle$. Upon receiving a transaction τ invoking contract A, a node computes $\gamma' = A \bmod m$ and routes τ to S-shard $\mathcal{S}_{\gamma'}$. The nodes in $\mathcal{S}_{\gamma'}$ will further route τ to its entry S-shard S_γ. During smart contract migration, all S-shards must also update their sub-tables to reflect the latest information. LightCross employs Kademlia [42] routing protocol to route transactions between nodes at the network level, like Ethereum does.

D. States Authentication

As mentioned in Section III-C, an executor has to request relevant states to drive the execution of a CSTx from involved S-shards. To ensure the communication is secure, a naïve approach lets the executor request the same state data from the majority of each S-shard. However, this will cause massive messages to be exchanged between the executor and S-shard. To avoid this, LightCross leverages the Merkle trees to implement a verifiable communication.

Let $\mathcal{E}$ be the executor that needs to request a state with key κ in contract A from S-shard $En(A)$. The detailed workflow goes through following steps.

1) $\mathcal{E}$ sends a request to a random node $\mathcal{N}_i$ in $En(A)$;
2) If $\mathcal{N}_i$ is correct, it responds with value d at $A[\kappa]$ and a Merkle proof $Cert$, which is used to authenticate d. In particular, the construction of $Cert$ follows the standard approach of *Merkle Patrica Tree* (MPT) in Ethereum[19, 43];
3) Once receiving d and $Cert$, $\mathcal{E}$ verifies d against $Cert$ following the Merkle verification;
4) If the verification fails, $\mathcal{E}$ changes to request another node in $\mathcal{S}_i$ from step 1) till one responds correctly.

A drawback of Merkle verification is that the executor $\mathcal{E}$ should obtain the root $root$ of MPT, included in every block header in the ledger of S-shard $En(A)$, in advance. This causes every executor to synchronize the last $root$s of Merkle trees from various S-shards. We address this issue through the *threshold signature*. An (n, t)-threshold signature on a message is a single, constant-sized aggregate signature that passes verification if and only if at least t out of the n participants sign the message correctly. Note that the verifier just needs to use a threshold public key of $En(A)$ to verify the threshold signature and does not need to know the identities of the t signers. In our

scenario, we assume there are $\mathbf{n}_\mathcal{S}$ nodes in a S-shard and at most $\mathbf{f}_\mathcal{S}$ nodes out of them are malicious, such that $\mathbf{n}_\mathcal{S} > 3\mathbf{f}_\mathcal{S}$. Then, the S-shard $En(A)$ creates a $(\mathbf{n}_\mathcal{S}, \mathbf{n}_\mathcal{S} - \mathbf{f}_\mathcal{S})$-threshold signature on $root$ at each block height. Specifically, every node broadcasts its signature on $root$, and then each node aggregates received $\mathbf{n}_\mathcal{S} - \mathbf{f}_\mathcal{S}$ signatures to a signature $AS(root)$. Since there are at most $\mathbf{f}_\mathcal{S}$ malicious nodes in a S-shard, as least $\mathbf{n}_\mathcal{S} - \mathbf{f}_\mathcal{S}$ honest nodes will broadcast their signatures correctly. This will not add extra overhead because these signatures can be attached to the messages of PBFT consensus.

Based on the threshold signature scheme, the workflow of the data request is adjusted into follows:

1) $\mathcal{E}$ sends a request to a node $\mathcal{N}_i$ in $En(A)$;
2) Upon receiving the request, $\mathcal{N}_i$ sends $AS(root)$ and $root$ to $\mathcal{E}$ alongside d and $Cert$;
3) $\mathcal{E}$ uses the threshold public key of $En(A)$, which is public known, to verify $AS(root)$;
4) $\mathcal{E}$ verifies d against $Cert$ and $root$;
5) If the verification fails, $\mathcal{E}$ retries again from step 1).

By this way, the $\mathcal{E}$ has no need to access a majority of nodes in $En(A)$, deeply reducing the amount of data to be transmitted. Moreover, $\mathcal{E}$ does not have to synchronize the block headers from various S-shards.

Compared with communication between executors and S-shards, the communication between executors and the R-shard seems simpler. The execution results generated by an executor are still verifiable through the attestation mechanism of TEE as described in Section III-C. Besides, as the data enclave at a executor has verified the input state data, R-shard $\mathcal{S}_m$ does not repeatedly check the read set of each transaction. Additionally, the communication between R-shard and S-shards, e.g., phases ④ and ⑤ in cross-shard commit protocol, is realized through a cluster-sending protocol proposed in [36]. This is the heaviest part of the communication mechanism in LightCross. Fortunately, LightCross amortizes the overhead of this part by processing multiple CSTxs in one batch.

E. Proof of Atomicity and Serializability

We first prove Theorem 1 that ensures the atomicity of CSTxs in LightCross, and then prove Theorem 2 based on Lemmas 1-3 to promise the serializability of all transactions.

Theorem 1 (Atomicity)**.** *If an S-shard in $shards(\tau)$ commits a CSTx τ, all the other S-shards in $shards(\tau)$ must commit transaction τ eventually.*

Proof. Let $k = |shards(\tau)| > 1$. If an S-shard commits τ, it means the R-shard has broadcast the final message $\langle bm_f \rangle$ (step ③ in Section III-D) to all the k involved S-shards. In this case, the S-shards in $rshard(\tau)$ must have locked all states in $R(\tau)$ read by τ (step ②). Before these $k-1$ S-shards commit τ (step ④), S-shards in $rshard(\tau)$ cannot unlocks states in $R(\tau)$, promising τ cannot be aborted. This is because τ only can be aborted when it reads state state values. The lock mechanism ensures the values of states read by τ still keep up-to-date till it is processed on each S-shards. On the other hand, the PBFT protocol ensures the liveness, i.e., all transactions must be processed eventually. As a result, τ will be processed by other $k-1$ S-shards but not aborted. This equally implies that τ will be committed by other S-shards in $shards(\tau)$ eventually. □

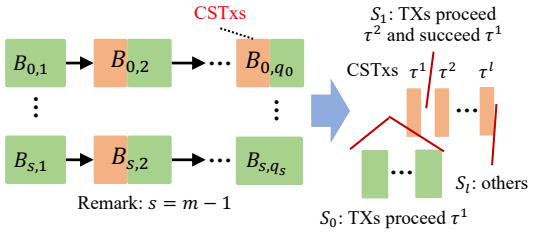


Fig. 9: Illustration for the proof of Theorem 2.

To prove the serializability, we need to give some notations and definitions. Let $B_{i,j}$ stand for the j-th block generated by S-shard $\mathcal{S}_i$. Obviously, it is impossible to serialize a CSTx τ that has not been committed by all S-shards in $shards(\tau)$, because its commitment has not completed yet. Therefore, we only can discuss the serializability of transactions that are committed by all involved S-shards. To this end, we introduce the conception of *committed block* as defined in Definition 1. Based on the committed block, we define the *committed chain & committed closure* as described in Definition 2. In a committed closure, every intra-shard or cross-shard transaction is committed correctly in $shards(\tau)$.

Definition 1 (Committed block)**.** *A block $B_{i,j}$, with no CSTx inside, is committed when $\mathcal{S}_i$ completes the executions for all transactions in $B_{i,j}$. Otherwise, $B_{i,j}$, including at least one CSTx, becomes committed if: 1) the execution for transactions in $B_{i,j}$ ends; and 2) for each CSTx $\tau \in B_{i,j}$, all S-shards in $shards(\tau)$ have committed it yet.*

Definition 2 (Committed chain & committed closure)**.** *A chain Φ ending with block $B_{i,j}$ is a **committed chain** if all of $B_{i,j}$'s ancestor blocks are committed. A set $\mathcal{C}$ of committed chains is called as a **committed closure** as if: 1) it contains m committed chains properly from m distinct S-shards; 2) for any CSTx τ in one committed chain, it exactly appears in $|shards(\tau)|$ different chains.*

Next, we give some properties of committed closure in Lemma 1 to Lemma 3, all of which illustrate the serializability of LightCross collectively.

Lemma 1. *In a committed closure $\mathcal{C}$, the effect of all transactions in $\mathcal{C}$ is equivalent to that of a serial execution in some order $\mathcal{L}$, which is termed as the serialization for $\mathcal{C}$.*

Proof. We directly construct the serialization $\mathcal{L}$ for $\mathcal{C}$ to prove this theorem. Assume the committed chains in $\mathcal{C}$ end with blocks $B_{0,q_0}, B_{1,q_1}, \ldots, B_{s,q_{m-1}}$, corresponding to m S-shards respectively, as shown in Fig. 9. Among these chains, we extract all CSTxs, denoted by $\langle \tau^1, \ldots, \tau^l \rangle$, where duplicated CSTxs just count once. Note that $\langle \tau^1, \ldots, \tau^l \rangle$ still conforms to the order established by R-shard. Then we fill other intra-shard

transactions into the gap between any two consecutive CSTxs to construct a complete serialization.

Let S_0 be a set of intra-shard transactions, each of which proceeds τ^1 in exactly one committed chain in $\mathcal{C}$. We assert that S_0 is serializable. The transactions in S_0 can be serialized in the format

$$\mathcal{L}_0 = \langle L_0, L_1, \ldots, L_{m-1} \rangle$$

where L_i is a ordered sequence of intra-shard transactions in S_0 processed by $\mathcal{S}_i$, and the transaction order in L_i is the same as in blocks generated by $\mathcal{S}_i$. We assert that the serialization for S_0 is correct because two transactions from two different L_i and L_j ($i \neq j$) do not conflict. Surely, we can serialize S_0 into other orders but reserve the partial order between transactions from the same committed chain. We construct the set S_1, where each transaction in S_1 succeeds τ^0 and proceeds τ^2 in some committed chain. S_1 can be serialized as $\mathcal{L}_1$ in the same way. We continuously construct every set S_k and serialize them properly until S_{l-1}. The remaining intra-shard transactions in $\mathcal{C}$ constitute the set S_l, which is serialized into $\mathcal{L}_l$.

Finally, the sequence $\langle \mathcal{L}_0, \tau^1, \mathcal{L}_1, \tau^2, \ldots, \mathcal{L}_{l-1}, \tau^l, \mathcal{L}_l \rangle$ is a correct serialization $\mathcal{L}$ for committed closure $\mathcal{C}$. In $\mathcal{C}$, we ensure each intra-shard or cross-shard transaction always reads the latest state values generated by proceeding transactions, keeping the same semantics as a serial execution. □

Then, we discuss the inclusion relationship between two committed closures as defined in Definition 3. Moreover, we prove the serializations for two committed closures with inclusion relationship are consistent and compatible in Lemma 2.

Definition 3 (Committed closure inclusion)**.** *A committed closure $\mathcal{C}_1$ is included in another closure $\mathcal{C}_2$, meaning that every chain Φ in $\mathcal{C}_1$ is the prefix of a chain Φ' in $\mathcal{C}_2$.*

Lemma 2. *Assume a committed closure $\mathcal{C}_1$ is included in another committed closure $\mathcal{C}_2$. For every serialization $\mathcal{L}$ for $\mathcal{C}_1$, we can always find a serialization $\mathcal{L}'$ for $\mathcal{C}_2$, such that $\mathcal{L}$ is the prefix of $\mathcal{L}'$.*

Proof. We prove this theorem by directly constructing the serialization $\mathcal{L}'$ for $\mathcal{C}_2$ with a prefix $\mathcal{L}$. First, since $\mathcal{C}_1$ is included in $\mathcal{C}_2$, every committed chain $\Phi \in \mathcal{C}_1$ is a prefix of one chain $\Phi' \in \mathcal{C}_2$. Then, we extract prefix Φ from Φ' and let $\mathcal{C}_3 = \{\Phi' - \Phi\}$ stand for remaining chains. We can assert that $\mathcal{C}_3$ is also a committed closure. Otherwise, at least one CSTx $\tau \in \mathcal{C}_2$ appears $< |shards(\tau)|$ times in $\mathcal{C}_3$. This indicates τ must appear in $\mathcal{C}_1$ because it appears exactly $|shards(\tau)|$ times in $\mathcal{C}_2$. However, since $\mathcal{C}_1$ is a committed closure, if τ appears in $\mathcal{C}_1$ one time, it will appear $|shards(\tau)|$ times in $\mathcal{C}_1$. Therefore, τ does not belong to $\mathcal{C}_3$, so that $\mathcal{C}_3$ is a committed closure.

Second, we serialize $\mathcal{C}_3$ in the same way adopted in the proof of Lemma 1, leading to a serialization $\mathcal{L}'$. Then, the concatenation $\overline{\mathcal{L}} = \langle \mathcal{L}, \mathcal{L}' \rangle$ is the final serialization. To verify $\overline{\mathcal{L}}$, we can check if $\overline{\mathcal{L}}$ satisfies the serialization construction in Lemma 1's proof. Summarized, the serialization $\overline{\mathcal{L}}$ is correct, and the lemma holds. □

Lemma 3. *Every committed transaction τ will eventually be included in a committed closure.*

Proof. We prove this theorem by contradiction. If τ is not included in any committed closure, finally, it means at least one S-shard $\mathcal{S}_i \in shards(\tau)$ will not commit τ forever. However, this violates Theorem 1, so the theorem holds. □

Theorem 2 (Serializability)**.** *If an S-shard commits a transaction τ (intra-shard or cross-shard), τ will be serialized properly.*

Proof. According to Lemma 1 and Lemma 3, we confirm that every transaction will eventually be appropriately serialized in LightCross. Besides, according to Lemma 2, the serialization for each transaction is durable and consistent with historical serializations. Then, this theorem holds. □